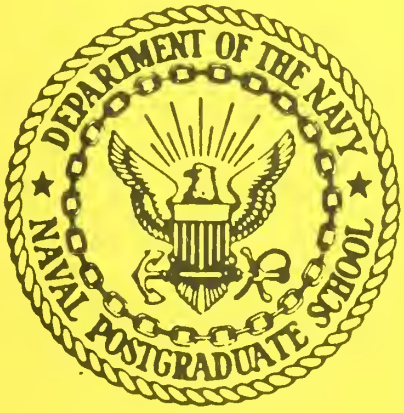
NPS52-86-028

NAVAL POSTGRADUATE SCHOOL

Monterey, California



PARAMETRIC REPRESENTATION AND POLYGONAL DECOMPOSITION OF CURVED SURFACES

MICHAEL J. ZYDA, ROBERT B. MCGHEE AND GARY W. TAYLOR

DECEMBER 1986

Prepared for:
Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL

Monterey, California

Rear Admiral R. C. Austin
Superintendent

D. A. Schrady
Provost

The work reported herein was supported by the U.S. Army Combat Developments Experimentation Center, Fort Ord, California and a grant from the Naval Ocean Systems Center, San Diego, California.

This report was prepared by:

VINCENT Y. LUM
Chairman
Department of Computer Science

KNEALE T. MARSHALL
Dean of Information and
Policy Science

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-86-028

2. GOVT ACCESSION NO.:

3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): PARAMETRIC REPRESENTATION AND POLYGONAL DECOMPOSITION OF CURVED SURFACES

5. TYPE OF REPORT & PERIOD COVERED:

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Michael J. Zyda, Robert B. McGhee and Gary W. Taylor

8. CONTRACT OR GRANT NUMBER(s):

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: N0001486WR4B123AC

11. CONTROLLING OFFICE NAME AND ADDRESS:

12. REPORT DATE: December 1986

13. NUMBER OF PAGES: 95

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): US Army Combat Dev. Experimentation Center, Fort Ord, CA; Naval Ocean Systems Center, San Diego, CA

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report): Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report):

18. SUPPLEMENTARY NOTES:

19. KEY WORDS (Continue on reverse side if necessary and identify by block number): parametric bicubic surface patches, graphics workstations, solid-filled surface patches

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

We present in this study a design and implementation for a set of software functions useful for constructing solid-filled parametric bicubic surface patches. Such a capability is not generally provided for in currently available, high-performance graphics workstation. Our implementation of this functionality is on one such workstation, the Silicon Graphics, Inc. IRIS. The capability for producing such solid-filled surface patches has promoted their use in a variety of in-house applications at the Naval Postgraduate School.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Parametric Representation and Polygonal Decomposition of Curved Surfaces

Gary W. Taylor, Robert B. McGhee and Michael J. Zyda *

Naval Postgraduate School,

Code 52, Dept. of Computer Science,

Monterey, California 93943

ABSTRACT

We present in this study a design and implementation for a set of software functions useful for constructing solid-filled parametric bicubic surface patches. Such a capability is not generally provided for in currently available, high-performance graphics workstations. Our implementation of this functionality is on one such workstation, the Silicon Graphics, Inc. IRIS. The capability for producing such solid-filled surface patches has promoted their use in a variety of in-house applications at the Naval Postgraduate School.

Key Words and Phrases: parametric bicubic surface patches, graphics workstations, solid-filled surface patches;

‡ This work was supported by the U.S. Army Combat Developments Experimentation Center, Fort Ord, California and a grant from the Naval Ocean Systems Center, San Diego (Ref. # N0001486WR4B123AC). This work was generated from Gary W. Taylor's Masters Thesis.

* Contact author.

TABLE OF CONTENTS

I. INTRODUCTION

A. BACKGROUND

A primary goal of a computer graphics system is to provide the user with different views of objects. Sometimes the objects that the user manipulates are simple in nature and can be constructed easily with the primitives provided by the graphics support package. However, in most applications areas the user is concerned with more complex objects. The display of three-dimensional surfaces is one such application area in computer graphics and is the area that this study explores.

It is the primary intent of this study to stimulate the reader's interest in the area of three-dimensional surface generation and display. To provide this stimulation, we combine the power of certain mathematical techniques and a high performance graphics environment to design and implement a set of functions that can be used to create, manipulate, and display three-dimensional solid-filled surfaces. Once developed, the reader will not only be able to use these functions to explore the design, representation, and rendering of such surfaces but also will be able to use these functions in other fields that can derive benefit from their use such as cartography, robotics, computer vision, and artificial intelligence.

B. PROJECT DESCRIPTION

1. Motivation

For developing graphics applications, the typical graphics environment provides the user with a variety of sophisticated, powerful tools and a high-level language. Almost inevitably, an application calls for something that is not provided for or if provided is not acceptable. The lack of solid-filled curved surface support in one particular high-performance graphics workstation prompted this study. By providing this capability we can create applications that enhance and expand what we are able to do and conceive of doing on these workstations.

2. Proposed Capabilities

To provide this capability for generating solid-filled parametric bicubic surface patches, we must look for a method that is simple, powerful, understandable, and implementable in software. One approach would be to start from scratch and develop a solution. This, however, is usually not a wise way to proceed. A solution derived in this manner is likely to have severe limitations such as being computationally inefficient, lacking robustness, and difficult to use. Another more prudent way to proceed is to find some basic mathematical techniques that can be applied to the problem in a form consistent with the existing graphics environment. Solutions developed in this manner are more easily accepted since their basis lies in proven mathematical techniques and generally meets the requirements we seek of being simple, powerful,

understandable and implementable. We chose to develop our new capability following the latter method - using pre-existing mathematics.

a. Parametric Bicubic Surface Construction

One of the primary capabilities needed when working with surfaces is to have an effective method for describing, manipulating and displaying them. For this, we base our work with surfaces on a mathematical model - the *parametric bicubic surface patch.* Choice of this method allows the user to rapidly develop a smooth surface and display it as a wireframe image through the specification of only a few points called *control points.*

b. Polygonal Parametric Bicubic Surface Decomposition

Another capability required is to extract, from the mathematical model, the information needed to construct a solid surface. For this, we need to be able to decompose any surface represented in parametric bicubic form into arbitrarily small polygons. These polygons can then be used in the construction of a polygon mesh. This capability permits the user to use standard or customized algorithms to manipulate the surface.

C. PROGRAMMING ENVIRONMENT

The IRIS Turbo 2400 Graphics Workstation, manufactured by Silicon Graphics, Inc., is the target programming environment for the design, development, and implementation of the parametric bicubic surface constructor and polygonal bicubic surface decomposition functions. The IRIS has special-

purpose hardware that is designed to replace less efficient software. The system supports real-time color graphics, the Unix operating system software, and the C programming language. A high resolution color monitor provides the output device for displaying the graphical output of the modeling and user defined functions.

In addition to the standard programming support provided by the UNIX operating system, the IRIS has an extensive collection of utility and graphics functions contained in a Graphics Library. This library provides the user high-level access to the hardware, enabling graphical objects to be easily manipulated as geometrical objects (points, lines, polygons, etc.) rather than pixels. A series of coordinate systems and mapping instructions also provides the user with the capability to define such objects in world coordinate space.

II. THREE DIMENSIONAL REPRESENTATION OF SURFACES

We stated in the previous chapter that an important application area in computer graphics is concerned with the three-dimensional representation of surfaces but we did not describe how one might do this. There are many ways to represent surfaces. The two most commonly used representations are: polygon meshes and parametric bicubic patches.

A. POLYGON MESHES

A *polygon mesh* is nothing more than a set of connected polygonally planar surfaces. There are several ways in which these planar polygons can be represented - explicit polygons, vertex list pointers, explicit edges, etc.. Each of these representations has its own advantages and disadvantages and various criteria can be applied to evaluate the representation. Criteria such as how much primary and secondary storage is available, how easy is it to identify the polygons sharing an edge, how difficult is it to display the mesh, name only a few commonly used for evaluation. Regardless of how the mesh is represented, there are many algorithms available for processing it. For example, algorithms have been developed to remove *hidden surfaces* from an object, while others can produce *lighting and shading* effects on the surface of the object.

Many objects that have planar features such as buildings, tables, and cabinets can easily be represented by a polygon mesh. However, polygon meshes are not limited to representing only planar objects. They can also be used to represent curved surfaces. To represent a *curved surface* with a polygon mesh, one creates an approximation to the surface by using arbitrarily small polygons. The smaller the polygons the better the approximation. However, certain difficulties arise when representing surfaces as polygon meshes. As one approximates the surface with smaller and smaller polygons, both space and execution time of the algorithms that process the mesh increase linearly.

B. PARAMETRIC BICUBIC PATCHES

Parametric bicubic patches are mathematical models of a surface and represent one of the simplest mathematical elements we can use to model an arbitrary surface. A patch can be defined as a curve bounded collection of points whose coordinates are given by a continuous, two-parameter, single valued function in the form

$$x = x(s,t)$$

$$y = y(s,t)$$

$$z = z(s,t)$$

where the parameters are restricted to

$$s,t \in [0,1].$$

Patches are used to describe the surface of an object in a piecewise manner much like the polygon mesh. That is, many patches are used to describe the surface of an object and the total surface is generated by displaying all of its individual patches.

Bicubic patches are most often used to generate line drawings of three-dimensional objects - often called *wireframe* representations. One benefit of using bicubic patches is that fewer bicubic patches than polygonal patches are needed to represent a curved surface to a given accuracy. On the other hand, the algorithms for working with bicubic surfaces are more complex than those for polygon meshes. In addition, wireframe representations can exhibit certain deficiencies. For example, objects that are represented by wireframes are often ambiguous. That is, you can look at the same object and get different visual interpretations. One reason for the different interpretations is the ability to *see through* a wireframe representation of an object. While not a problem for some applications, it can be a serious problem for others.

III. PARAMETRIC CUBIC CURVES AND SURFACES

A. GENERAL

The method that we use to generate surfaces is based on using parametric cubic curves so it is helpful to review the mathematical basis for these curves. A *parametric cubic curve* has the property that x, y, and z can be defined as third-order polynomials for some variable t [Ref. 1:pp. 34] :

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x,$$
$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y,$$
$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z.$$

These equations are known as the *algebraic form* of a parametric cubic curve. In this form, we can identify a unique set of 12 constant algebraic coefficients. These algebraic coefficients determine a unique parametric cubic curve; they determine the size and shape of the curve and its position in three-dimensional space. Two curves of the same shape have different algebraic coefficients if they occupy different positions in three-dimensional space. Because we want to deal with a finite segments of the curve, we limit the range of the parameter, without loss of generality, to $0 \leq t \leq 1$. We call these finite pieces curve segments. A *curve segment* is nothing more than a point-bounded collection of points. In our case the points are bounded at t=0 and t=1. Each equation in the algebraic form can be expressed as a vector product as follows:

$$x(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix} = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} C_x = TC_x.$$

Using the vector product form is usually more convenient as it separates the distinct parameters of the parametric equation into unknown coefficients of $x(t)$ and the parameter t that we wish to manipulate. Here T is the row vector of powers of t, while C_x is the column vector of coefficients of $x(t)$. Similarly the parametric equations for $y(t)$ and $z(t)$ can be written as $y(t) = TC_y$ and $z(t) = TC_z$. By varying the parameter t from 0 to 1 in each equation we define the curve segment.

Arbitrarily assigning values to these unknown coefficients results in defining a curve in three-dimensional space. However, it is not easy to determine the properties of this curve. What we wish to do is establish some *constraints* on these coefficients. We want the curves we generate to have some predictable properties. To solve the equations for these unknown algebraic coefficients, we establish a set of constraints, thereby defining a unique cubic curve with predictable properties. To illustrate this process we look at some example cubic curves for which the constraints are well-known.

B. CUBIC CURVE EXAMPLES

1. Hermite Curve

The *Hermite cubic curve* is determined from its endpoints (P_1, P_2) and endpoint tangents (R_1, R_2). In the literature [Ref. 2:pp. 516-519] [Ref. 3:pp. 123-

129], we find the *geometric form* of the Hermite curve to be

$$Q_{h(t)} = TM_hG_h.$$

In geometric form, T is the row vector of the powers of the parametric variable t, M_h is the basis matrix, and G_h is the geometry vector. A basis matrix refers to a constraint procedure that is embodied in matrix form and a geometry vector contains the control points used to guide the curve.

In this particular case, the *Hermite basis matrix* (M_h) is

$$\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

and the *Hermite geometry vector* is

$$\begin{bmatrix} P_1 \\ P_2 \\ R_1 \\ R_2 \end{bmatrix}.$$

Now using the above formulation, given two points and their tangents, we can evaluate $x(t)$, $y(t)$, and $z(t)$ for $0 \leqslant t \leqslant 1$ and find all points on the Hermite form of the cubic curve from P_1 to P_2 with starting tangent vector R_1 and ending tangent vector R_2. It is through these constraints (M_h) that the control points (G_h) control the parametric equations and produce an equation that can generate a discretely sampled curve segment in three-dimensional space.

If we take the product $\mathbf{TM_h}$, we have

$$\mathbf{TM_h} = \left[(2t^3-3t^2+1) \ (-2t^3+3t^2) \ (t^3-2t^2+t) \ (t^3-t^2) \right].$$

These four functions of t in the product $\mathbf{TM_h}$ are often called *blending functions* [Ref 1:pp. 48-52]. As the name implies, they blend the effects or contributions of the endpoints and tangent vectors to produce the intermediate point coordinate values over the domain of t.

2. Bezier Curve

The defining form for a *Bezier cubic curve* is similar to the Hermite form. The difference is in the definition of the endpoint tangent vectors. The Bezier form uses four points ($\mathbf{P_1}, \mathbf{P_2}, \mathbf{P_3}, \mathbf{P_4}$) instead of 2 points and 2 tangent vectors. The tangent vectors at the endpoints in Bezier form are determined by the line segments $\mathbf{P_1P_2}$ and $\mathbf{P_3P_4}$. The Bezier cubic curve passes through the first and fourth control points ($\mathbf{P_1}$ and $\mathbf{P_4}$) and uses the second and third points ($\mathbf{P_2}$ and $\mathbf{P_3}$) to determine the shape of the curve. [Ref. 1:pp. 113-125] [Ref. 2:pp. 519-521].

The geometric form of the Bezier curve is

$$\mathbf{Q_b}(t) = \mathbf{TM_bG_b}$$

where the *Bezier basis matrix* ($\mathbf{M_b}$) is

$$\mathbf{M_b} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

and the *Bezier geometry vector* is

$$\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}.$$

The Bezier form of the cubic curve is more widely used in computer graphics than the Hermite form. A primary reason for its popularity is that the geometry matrix of four points (G_b) is more intuitive for an interactive user. The user has only to manipulate the four points and does not have to specify the tangent vectors. It is usually easier for a person to think about manipulating points rather than trying to manipulate points and tangent vectors.

3. Other Useful Cubic Curves

The Hermite and Bezier cubic curves are not the only forms of cubic curve that are available. Two others are the Cardinal Spline and the B-Spline.

a. Cardinal Spline

The *Cardinal Spline curve* passes through the two interior control points (P_2 and P_3) and uses the points P_1 and P_4 to define the shape of the curve [Ref. 4:p. 11-4]. The geometric form of the Cardinal curve is

$$Q_c(t) = TM_cG_c$$

where the *Cardinal basis matrix* (M_c) is

$$\begin{bmatrix} -a & 2-a & a-2 & a \\ 2a & a-3 & 3-2a & -a \\ -a & 0 & a & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

and the *Cardinal geometry vector* is

$$\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}.$$

The scalar coefficient a in the Cardinal basis matrix must be positive and determines the length of the tangent vector at point P_2 and P_3.

b. B-Spline

The geometric form of the *B-Spline curve* is

$$Q_{bs}(t) = TM_{bs}G_{bs}$$

where the *B-Spline basis matrix* (M_{bs}) is

$$\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

and the *B-Spline geometry vector* is

$$\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}.$$

In general, the B-Spline curve does not pass through any control points but is continuous and also has continuity of tangent vectors and of curvature (that is, first and second derivatives are continuous at the endpoints). The Hermite and Bezier forms have only first-derivative continuity at the endpoints, but do pass through control points [Ref. 1:pp. 125-146] [Ref. 2:pp. 521-523].

C. DEFINING SURFACES

By adding a new parameter s and additional algebraic coefficients to the cubic curves in the previous section, we can define the algebraic form of a bicubic surface patch [Ref. 1:pp. 156] as

$$p(s,t) = \sum_{I=0}^{I=3} \sum_{J=0}^{J=3} a_{IJ} s^I t^J$$

with the restriction on the parametric variables to

$$s,t \in [0,1].$$

By varying both parameters from 0 to 1 in each equation, we define all points on the surface patch. Assigning one parameter a constant value and varying the other, results in a cubic curve.

Expanding the above equation in terms of $x(s,t)$ and noting that the terms for $y(s,t)$ and $z(s,t)$ are similar we have

$$x(s,t) = a_{33}s^3t^3 + a_{32}s^3t^2 + a_{31}s^3t + a_{30}s^3 + a_{23}s^2t^3 + a_{22}s^2t^2 + a_{21}s^2t + a_{20}s^2$$
$$+ a_{13}st^3 + a_{12}st^2 + a_{11}st + a_{10}s + a_{03}t^3 + a_{02}t^2 + a_{01}t + a_{00}.$$

Written in vector product form

$$x(s,t) = SC_xT^T$$

where

$$S = \begin{bmatrix} s^3 & s^2 & s & 1 \end{bmatrix},$$

$$T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix},$$

$$C_x = \begin{bmatrix} a_{33} & a_{32} & a_{31} & a_{30} \\ a_{23} & a_{22} & a_{21} & a_{20} \\ a_{13} & a_{12} & a_{11} & a_{10} \\ a_{03} & a_{02} & a_{01} & a_{00} \end{bmatrix}$$

and T^T is the transpose of the matrix T .

From these equations we can see that there are 48 degrees of freedom or algebraic coefficients that we must specify. Like the cubic curve, a change in any one of these coefficients defines a different surface.

The complete algebraic manipulation of the equations to arrive at the following equation is similar to that of the curve process described in the previous section. For the Bezier surface patch, the geometric form of the equation is:

$$x(s,t) = SM_bQ_xM_b^TT^T$$

where M_b is the same M_b as in the Bezier curve equation, M_b^T is its transpose, and

Q_x is the **x** component of the sixteen control points of a surface patch. The matrix Q_x is

$$Q_x = \begin{bmatrix} P_{1_x} & P_{5_x} & P_{9_x} & P_{13_x} \\ P_{2_x} & P_{6_x} & P_{10_x} & P_{14_x} \\ P_{3_x} & P_{7_x} & P_{11_x} & P_{15_x} \\ P_{4_x} & P_{8_x} & P_{12_x} & P_{16_x} \end{bmatrix}$$

and similarly for Q_y and Q_z.

Since we must provide three 4x4 matrices, one for each of component **x**, **y**, and **z**, it can be seen that we have specified the 48 degrees of freedom as in the algebraic form.

As can be seen by the above equations, a bicubic surface patch can be defined by a set of 16 control points and a basis matrix. By manipulating the control points, we change the shape of the surface as constrained by the basis matrix. We take this knowledge with us as we design our functions in the next chapter.

IV. DESIGN AND IMPLEMENTATION OF OUR SURFACE FUNCTIONS

A. OVERVIEW

Having the mathematics developed in the previous chapter does us no good if we can not to put it to use. In our case, this means being able to draw a parametric bicubic surface. It is at this point that we begin to see how the mathematics can be combined with the power of the computer and the graphics workstation.

Up to this point, we have dealt with two forms of the parametric cubic curve and parametric bicubic surface - the algebraic form and the geometric form. The question now is which one shall we work with?

Deciding what form to use depends largely on the application. If we are given or know the algebraic equations of the curve, then the reasonable choice is the algebraic form. If we plan to do surface fitting of data or interactive design the choice is the geometric. We choose to use the geometric form. Our primary reason for choosing it is that the geometric form offers us a greater insight into the control and behavior of curves and surfaces than is otherwise available with the classical algebraic formulation. It should be noted, however, that it is possible, through mathematical manipulation, to convert from one form to the other [Ref. 1:pp. 164].

B. METHODOLOGY

How one proceeds to generate a surface impacts usefulness, flexibility, and the ability to understand. As we stated in the first chapter, the built-in functions provided by a graphics environment are not always exactly what we want. This is the case in the IRIS graphics environment. Although the IRIS provides support for bicubic surface patches, it does not support surface patch decomposition. What we want is the capability to do both. We also want this capability without sacrificing what a user already knows about how the IRIS supports bicubic surface patches. To achieve this, we have developed a set of *parallel routines* [Appendix A] that provide nearly all the functionality as the standard IRIS functions while at the same time providing the user with extended support via three additional functions. These extended functions allow the user to have access to the triangular polygons that our new functions generate during the construction of the bicubic surface patch. That is, the IRIS user is able to use the new functions in the same way as he would use the standard functions by substituting the names of the new routines in place of the standard IRIS routines. If, however, the user wishes to be able to have access to the individual triangular polygons that make up the surface, he has only three additional routines to learn.

C. PARALLELING THE IRIS SUPPORTED FUNCTIONS

The IRIS graphics environment has available five functions for defining and generating parametric bicubic surface patches. Those five functions are:

- ***defbasis*** defines a basis matrix
- ***patchbasis*** sets the current basis matrices for both the s and the t parametric direction
- ***patchcurves*** sets the number of curves used to represent a patch
- ***patchprecision*** sets the precision at which the curves are drawn
- ***patch*** draws the surface patch.

A complete description of the functions and their arguments can be found in the IRIS Users Manual [Ref. 4].

To ease the pain of learning new functions, our new functions are syntactically identical to the standard IRIS functions with the exception that the new function names are the standard function names prefixed by the letter ***n***. These parallel functions are:

- ***ndefbasis***
- ***npatchbasis***
- ***npatchcurves***
- ***npatchprecision***
- ***npatch***.

The usage and the arguments of these parallel functions remain the same as the standard IRIS functions. The only difference that the user notices is that the wireframe drawn looks like a triangular mesh instead of the typical wireframe and that the function ***npatchprecision*** has no effect on the displayed image.

While these routines seem to do what the old routines do, they are more powerful because they provide special *extensions* to the user. These extensions provide the capability to manipulate the surface patch as individual polygons.

D. TRIANGULAR DECOMPOSITION OF SURFACE PATCHES

The extensions mentioned in the previous section are available to the user by using three additional routines:

- ***Set_User_Routine_for_npatch*** provides an *intercept function* for handling the triangular polygons generated in the surface decomposition

- ***User_Routine*** provides the user a way to turn the intercept function on and off

- ***Set_Default_Routine_for_npatch*** allows the user to return to the system defined intercept function.

There is one argument to the function ***Set_User_Routine_for_npatch***. This argument is the name of a user-defined function that expects to receive a 3x3 array. This 3x3 array contains the three vertices of a triangle where each vertex is made up of an x, y, and z coordinate. The function ***User_Routine*** expects one argument also. If this argument is zero, then the intercept function is turned off; i.e., the user's program cannot intercept the triangles composing the surface. Otherwise the function is activated allowing the user's program to intercept the triangles comprising the surface. The function ***Set_Default_Routine_for_npatch*** does not expect any arguments.

Using these functions, the user's program has access to and can manipulate the individual triangular components of the surface patch. For example, an individual surface patch can be decomposed into triangular polygons and then via the user intercept function, each polygon can be subjected to an illumination model that produces a realistic looking surface in three-dimensional space.

The surface patch is decomposed into triangular polygons one at a time. For the user's program to intercept these polygons. the program must have specified an intercept function via the ***Set_User_Routine_for_npatch*** and must have activated it via ***User_Routine.*** Then, as each polygon is generated, the user's function can process them in any way desired. They can be stored, manipulated, altered, etc.. It is the user's program that determines what to do with them. This feature provides a tremendous amount of flexibility, creativity, and applicability above what is currently available in the standard IRIS graphics environment support of surfaces.

E. GENERAL GUIDELINES FOR USAGE

To prevent any unnecessary problems in using our new functions, we need to establish a basic set of guidelines or sequences of events that should be followed. If the user does not want to use the special extensions, i.e. ***Set_User_Routine_for_npatch*** and ***User_Routine,*** then a modified version of the standard IRIS setup steps for using surface patches can be followed. These steps are:

- define the appropriate curve bases using the ***ndefbasis*** function;
- select a basis for the s and t parametric directions using the ***npatchbasis*** function;
- select the number of curve segments to be drawn in each parametric direction using the ***npatchcurves*** function;
- draw the surface by using the ***npatch*** function.

The only change to the standard IRIS setup is that it is not necessary to use

the ***npatchprecision*** function. This function does not effect the displayed image and its only purpose is to maintain consistency with the standard IRIS functions.

If the user wishes to use the extensions via the ***Set_User_Routine_for_npatch*** and the ***User_Routine*** functions, then the following steps must added:

- An intercept function must be declared and defined before calling the ***Set_User_Routine_for_npatch*** function. This intercept function must be declared as a function returning an integer value (even though it is not used) and must be defined as receiving a 3x3 matrix of floating point numbers, where each row contains one set of `x`, `y` and `z` coordinates of an intercepted triangles vertex. The name of this intercept function will be the argument given to the ***Set_User_Routine_for_npatch*** function;

- Activating/deactivating the intercept function via the ***User_Routine*** function can be performed any time after the above step has been completed.

By using these guidelines, a user should not have any difficulty in using these functions. As we will show in the next chapter, these functions are easy to use, efficient, and can provide some impressive results when a carefully chosen intercept function is used.

V. USAGE AND PERFORMANCE

Having taken a brief tour of the functions we designed in the previous chapter, we need to provide some concrete examples of their usage, performance levels, and limitations.

A. SAMPLE PROGRAMS

Appendix B gives the listing for four sample programs using the new functions. Each program illustrates how the new functions can be integrated into the IRIS graphics programming environment. Program #1 draws 2 surface patches in wireframe representation. One surface is drawn using the standard IRIS functions while the other is drawn using the parallel functions. When this program is run the user notices the different appearance of the wireframe surface patch drawn with the new functions. It has the triangular mesh appearance described in the previous chapter. Program #2 shows how a user-defined intercept function can be used via the three extension functions we have designed. This program intercepts the triangles generated during the patches decomposition and puts them into an IRIS graphical object. Program #3 shows how the user-defined intercept function can be dynamically changed during execution and Program #4 shows how a well chosen intercept function can be used to produce realistic lighting of a surface.

B. PERFORMANCE COMPARISONS

Knowing that our functions work, we would like to know how efficient they are. The way that we approach this question is to compare our new functions to the standard IRIS functions. Because we have designed our functions to be substitutable as a set for the standard functions, we do not have any problems in testing the relative performance of the sets. However, two words of caution are in order before comparing these two sets of functions. First, the IRIS implements many of its graphics primitives via special purpose hardware. This is the case with the function ***patch***. Therefore, its parallel function ***npatch***, which is implemented in software is not as fast. Second, the new function ***npatchprecision*** does not affect the computation whereas the IRIS function ***patchprecision*** does affect the computations. Keeping these points in mind, we have developed a simple benchmark program, listed in Appendix C, that we use to draw a wireframe representation of a surface patch 100 times. By executing this program 10 times and getting the average times, we can get an idea of the performance of the parallel set of surface patch functions as compared to the standard IRIS surface patch functions.

The way that we measure performance of a particular program is to use the UNIX **time** command. The time command returns, on program completion, the time in seconds for *system time*, *user time*, and *elapsed time*.

The benchmark program was executed in two different modes. In the first mode, the program was executed without the assistance of the IRIS's *floating*

point accelerator (FFP) while in the second mode, the program was executed with the FFP. The results indicate that the standard IRIS functions are 350% faster without the FFP and 260% faster with the FFP. As expected, the new functions are slower, however considering that they were not designed to replace the IRIS functions, these results are good. Normally, one can expect an order of magnitude increase in performance when special hardware is used.

C. LIMITATIONS

Nothing that can be developed is without limitations. The reader should recall that it was certain limitations of the existing IRIS system that motivated this study. The functions we have designed and have implemented have allowed us to overcome certain limitations in the IRIS graphics environment. At the same time, these functions have their own limitations.

The primary limitation of our parallel functions is speed. While these functions have been carefully implemented using efficient algorithms and data structures, they are not as fast as using special purpose hardware. Another limitation deals with the use of memory. The ***npatch*** function allocates memory to save each point on the surface patch. The number of points that are generated are proportional to the product of the desired number of curve segments in the s and t parametric directions. For example, to draw a surface patch with 10 curves in the s direction and 10 curves in the t directions requires at least enough memory to store 300 floating point numbers (one for each x, y, and z component).

To draw a 100x100 surface patch requires enough memory to store 30,000 floating point numbers. Assuming a floating point number requires 4 bytes, the 10x10 patch requires 1.17 Kilobytes of memory, while the 100x100 patch requires 117.1 Kilobytes of memory.

VI. RECOMMENDATIONS AND CONCLUSIONS

A. DIRECTIONS FOR FURTHER STUDY

Bicubic surface display and generation is an area of research in computer graphics that is exciting and important. Since the development of high-performance graphics systems, the demand for realism and real-time has increased significantly. Consequently, the need is great for continued creativity and exploration in the area.

1. Development of Application Programs

The power of these parallel surface functions we have created can only be derived through the use of the intercept functions. Whether they will be used to experiment with lighting and shading models or applied to fractal geometry can only be answered by time. However, it is through creative experimentation that these questions can be answered. Some areas for further work are:

- **Surface-fitting sampled data**
  Surface-fitting is the process of constructing a representation to model the surface of an object based on a fairly large number of given data points. By taking these points and chosing an appropriate set of surface constraints, one can accurately reconstruct the surface. For example, during this work, we were given a set of digitized **x**, **y** and **z** coordinates for a human head. By successively extracting control points from the data, we were able to reproduce and display the head quite accurately.

- **Data Reduction**
  In many instances it is possible to reduce the amount of data needed to properly reconstruct a surface. For example, consider geographical terrain. Terrain that is relatively *flat* can be reconstructed with fewer surface patches that can *mountainous* terrain. The problem is that most terrain is sampled at discrete intervals, such as every 100 meters, whether it is flat or not. By applying some form of an *Adaptive Subdivision Algorithm* [Ref. 5] one can reduce the amount of primary and secondary storage while at the same time provide increased performance for display.

- **Lighting Models**
  Because the user can intercept individual polygons comprising the surface, it is possible to subject each polygon to a lighting model. While we have provided a simple example of this, more sophisticated lighting models could be easily integrated through these parallel functions.

- **Realistic 3-D Objects**
  The surfaces of many vehicles such as automobiles, aircraft, and ships can be constructed with bicubic surface patches. For example, constructing an object with surface patches and applying a lighting and shading model, one could develop an ship identification training system. Such a training system would be a valuable asset in military training environments, allowing the trainee to view a particular class of ship from any viewing angle.

2. Improvement of Performance

Real-time computer graphics requires efficient algorithms and data structures. While these functions were coded to be as efficient as possible, while preserving understandability, there is always room for improvement. One suggestion we have is to contact the developers of the IRIS graphics package for insights into improving our packages performance. Such contact may provide access to low-level graphic system routines and techniques that could dramatically improve performance.

B. CONCLUSIONS

This study introduced the reader to the world of parametric bicubic surfaces. To do this, we provided some necessary definitions, terminology and mathematics. We also designed and implemented a set of software functions that take advantage of the information and given them to the reader for experimentation. The benefit that can be derived from the use of these functions can only be determined by the passage of time.

APPENDIX A – FUNCTION SPECIFICATIONS

NAME

ndefbasis - defines a basis matrix

SPECIFICATION

ndefbasis(id, mat)
long id;
Matrix mat;

DESCRIPTION

ndefbasis allows the user to define basis matrices for use in the generation of patches. *matrix* is saved and is associated with *id*. *id* may then be used in subsequent calls to **npatchbasis**.

NAME

npatchbasis - sets current basis matrices

SPECIFICATION

npatchbasis(sid, tid)
long sid, tid;

DESCRIPTION

npatchbasis sets the current basis matrices (defined by **ndefbasis**) for both the *s* and *t* parametric directions of a surface patch. The current *s* and *t* bases are used when the **npatch** command is issued.

NAME

npatchcurves - sets number of curves used to represent a patch

SPECIFICATION

npatchcurves(scurves, tcurves)
long scurves, tcurves;

DESCRIPTION

npatchcurves sets the current number of *s* and *t* curves used to represent a patch as a wireframe.

NAME

npatchprecision - is a null function.

SPECIFICATION

npatchprecision(ssegments, tsegments)
long ssegments, tsegments;

DESCRIPTION

npatchprecision has no functionality at the current time. It is used to maintain consistency with the standard IRIS function **patchprecision.**

NAME

npatch - draws a surface patch

SPECIFICATION

npatch(geomx, geomy, geomz)
Matrix geomx, geomy, geomz;

DESCRIPTION

npatch draws a surface patch using the current **npatchbasis** and **npatchcurves**. The shape of the patch is determined by the control points specified in *geomx*, *geomy*, and *geomz*.

NAME

Set_User_Routine_for_npatch - allows the user to specify an intercept function

SPECIFICATION

Set_User_Routine_for_npatch(fname)
int (*fname)();

DESCRIPTION

Set_User_Routine_for_npatch allows the user to set up a function that is capable of intercepting triangular polygons generated during the decomposition of a surface patch. The number of polygons generated is (*(scurves - 1) * (tcurves - 1) * 2.*).

NAME

User_Routine - allows the user to turn the intercept function on and off

SPECIFICATION

User_Routine(boolean)
int boolean;

DESCRIPTION

User_Routine acts like a switch allowing a user-defined intercept function to be turned on and off. By assigning ***boolean*** the value 0 the intercept function is turned off. Integer values other than 0 cause the intercept function to be turned on.

NAME

Set_Default_Routine_for_npatch - resets the intercept function to a system defined default

SPECIFICATION

Set_Default_Routine_for_npatch()

DESCRIPTION

Set_Default_Routine_for_npatch enables the user to choose the system defined intercept function **poly(3, Triangle)**.

APPENDIX B -- DEMONSTRATION PROGRAMS

```
/* This is file "basis.h" */
#define  HERMITE   0
#define  BEZIER    1
#define  CARDINAL  2
#define  BSPLINE   3

/* the HERMITE BASIS MATRIX */
Matrix hermitematrix = {
    {  2.0, -2.0, 1.0, 1.0  },
    { -3.0, 3.0, -2.0, -1.0 },
    { 0.0, 0.0, 1.0, 0.0    },
    { 1.0, 0.0, 0.0, 0.0    }
};

/* the CARDINAL BASIS MATRIX */
Matrix cardinalmatrix = {
    { -0.5, 1.5, -1.5, 0.5  },
    { 1.0 , -2.5, 2.0, -0.5 },
    { -0.5, 0.0, 0.5, 0.0   },
    { 0.0, 1.0, 0.0. 0.0    }
};

/* the BEZIER BASIS MATRIX */
Matrix beziermatrix = {
    { -1.0, 3.0, -3.0, 1.0  },
    { 3.0, -6.0, 3.0, 0.0   },
    { -3.0, 3.0, 0.0. 0.0   },
    { 1.0, 0.0, 0.0, 0.0    }
};

/* the B-SPLINE BASIS MATRIX */
Matrix bsplinematrix = {
    { -1.0/6.0, 3.0/6.0, -3.0/6.0, 1.0/6.0  },
    { 3.0/6.0, -6.0/6.0, 3.0/6.0, 0.0   },
    { -3.0/6.0,    0.0, 3.0/6.0, 0.0    },
    { 1.0/6.0,  4.0/6.0, 1.0/6.0, 0.0   }
};
```

```
/* This is file "geom.h" */

/* set up the geometry matrix of x coordinates */
Coord geomx[4][4] = {
    { 0.0,    100.0, 200.0, 300.0 },
    { 0.0,    100.0, 200.0, 300.0 },
    { 1000.0, 900.0, 800.0, 700.0 },
    { 1000.0, 900.0, 800.0, 700.0 }
};

/* set up the geometry matrix of y coordinates */
Coord geomy[4][4] = {
    { 400.0, 500.0. 600.0, 700.0 },
    { 0.0,   200.0, 400.0, 600.0 },
    { 0.0,   200.0, 400.0, 600.0 },
    { 400.0, 500.0, 600.0, 700.0 }
};

/* set up the geometry matrix of z coordinates */
Coord geomz[4][4] = {
    { 0.0. 200.0, 400.0, 800.0 },
    { 0.0. 200.0, 400.0, 800.0 },
    { 0.0, 200.0, 400.0, 800.0 },
    { 0.0, 200.0, 400.0, 800.0 }
};
```

```
/*****************************************************************
                Program #1

  This program displays two wireframe images of the same
  surface patch.  The patch drawn in the color YELLOW is
  produced by the standard IRIS patch functions and the
  patch drawn in the color RED is produced by the parallel
  functions we have developed.

  One notices that the patch drawn via the parallel functions
  has a triangular mesh appearance and that the call to the
  npatchprecision does not affect the displayed image as
  does the standard IRIS patchprecision function..

*****************************************************************/

#include "gl.h"     /* IRIS graphics library */

#include "basis.h"
#include "geom.h"

#define  S_CURVES  10
#define  T_CURVES  10

main()
{
    /* Loop variables */
    int p1, p2;

    /* initialize the graphics system */
    ginit();
    doublebuffer();
    gconfig();
    cursoff();

    /* set up the viewing parameters */
    ortho(0.0, 1023.0, 0.0, 1023.0, -1023.0, 1023.0);
    viewport(0, 1023, 0, 767);

    /* clear the graphics screen to BLACK */
    color(BLACK);
    clear();

    /* Associate an id number with a basis matrix */
    defbasis(BEZIER, beziermatrix);
    defbasis(CARDINAL, cardinalmatrix);
    defbasis(BSPLINE, bsplinematrix);

    ndefbasis(BEZIER, beziermatrix);
    ndefbasis(CARDINAL, cardinalmatrix);
    ndefbasis(BSPLINE, bsplinematrix);
```

```
    /** Specify how many curves in each
       parametric direction. **/
    patchcurves(S_CURVES , T_CURVES);
    npatchcurves(S_CURVES , T_CURVES);

    /** Make the basis matrices different
       for each parametric direction. **/
    patchbasis(BEZIER, CARDINAL);
    npatchbasis(BEZIER, CARDINAL);

    /** Cycle through the patch changing the
       precision that the individual curves
       comprising the patch are drawn. **/
    for(p1 = 10, p2 = 100; p1 < 100; p1 += 5, p2 -= 5) {

        /* Draw the image via the IRIS functions */
        viewport(0, 511, 0, 767);
        color(BLACK);
        clear();
        color(YELLOW);
        patchprecision(p1, p2);
        patch(geomx, geomy, geomz);

        /* Draw the image via the parallel functions */
        viewport(512, 1023, 0, 767);
        color(BLACK);
        clear();
        color(RED);
        npatchprecision(p1, p2);
        npatch(geomx, geomy, geomz);

        /* display the wireframe images */
        swapbuffers();
        sleep(1);

        /* clear the screen */
        color(BLACK);
        clear();
    }
    /* clear the graphics screen and exit */
    color(BLACK);
    clear();
    gexit();
}
```

```
/*****************************************************************
                Program #2

  This program illustrates how the triangles formed during
  a surface patch decomposition can be put into an IRIS
  graphical object and subsequently displayed.

*****************************************************************/

#include "gl.h"     /* IRIS graphics library */

#include "basis.h"
#include "geom.h"

#define  ON        1
#define  S_CURVES  10
#define  T_CURVES  10

/* Where we put our intercepted triangles */
Object intercepted_object;

main()
{
    /* declare the intercept function */
    int intercept_function();

    /* initialize the graphics system */
    ginit();
    doublebuffer();
    gconfig();
    cursoff();

    /* Make the intital object. */
    makeobj(intercepted_object = genobj());
    closeobj();

    /* set up the viewing parameters */
    ortho(0.0, 1023.0, 0.0, 1023.0, -1023.0, 1023.0);
    viewport(0, 1023, 0, 767);

    /* clear the graphics screen to BLACK */
    color(BLACK);
    clear();

    /* Associate an id number with a basis matrix */
    ndefbasis(BEZIER, beziermatrix);
    ndefbasis(CARDINAL, cardinalmatrix);
    ndefbasis(BSPLINE, bsplinematrix);

    /** Specify how many curves in each
       parametric direction. **/
    npatchcurves(S_CURVES , T_CURVES);
```

```
    /** Make the basis matrices different
        for each parametric direction.**/
    npatchbasis(BEZIER, BSPLINE);

    /** Set up an intercept function to grab
        the triangles from the surface patch
        and turn it on. **/
    Set_User_Routine_for_npatch(intercept_function);
    User_Routine(ON);

    /* Call the npatch function. */
    npatch(geomx, geomy, geomz);

    /* Display the surface patch. */
    color(YELLOW);
    callobj(intercepted_object);
    swapbuffers();
    sleep(10);

    /* clear the graphics screen and exit */
    color(BLACK);
    clear();
    gexit();

}


/*************************************************************
  This is the user-defined intercept function that
  handles the individual triangles generated
  during the decomposition of a surface patch.
*************************************************************/
int intercept_function(triangle)
float triangle[3][3];
{
    /* Open up the object and put in the triangles */
    editobj(intercepted_object);
    move(triangle[0][0], triangle[0][1], triangle[0][2]);
    draw(triangle[1][0], triangle[1][1], triangle[1][2]);
    draw(triangle[2][0], triangle[2][1], triangle[2][2]);
    draw(triangle[0][0], triangle[0][1], triangle[0][2]);
    closeobj();
}
```

```
/*****************************************************************
            Program #3

 This programs illustrates how the intercept function
 can be changed as the program runs.

*****************************************************************/

#include "gl.h"     /* IRIS graphics library */

#include "basis.h"
#include "geom.h"

/** Define a new type that is a pointer to a function
   that returns an integer. **/
typedef  int (*Function_pointer)() ;

#define  ON       1
#define  S_CURVES  10
#define  T_CURVES  10

main()
{
    /* Loop variable */
    int count;

    /* Declare an array of pointers to functions */
    Function_pointer intercept_functions[4];

    /* declare the intercept functions */
    int intercept_function1();
    int intercept_function2();
    int intercept_function3();
    int intercept_function4();

    /** Initialize the array of pointers to the
       intercept functions. **/
    intercept_functions[0] = intercept_function1;
    intercept_functions[1] = intercept_function2;
    intercept_functions[2] = intercept_function3;
    intercept_functions[3] = intercept_function4;

    /* initialize the graphics system */
    ginit();
    doublebuffer();
    gconfig();
    cursoff();

    /* set up the viewing parameters */
    ortho(0.0, 1023.0, 0.0, 1023.0, -1023.0, 1023.0);
    viewport(0, 1023, 0, 767);
```

```
    /* clear the graphics screen to BLACK */
    color(BLACK);
    clear();

    /* Associate an id number with a basis matrix */
    ndefbasis(BEZIER, beziermatrix);
    ndefbasis(CARDINAL, cardinalmatrix);
    ndefbasis(BSPLINE, bsplinematrix);

    /** Specify how many curves in each
       parametric direction. **/
    npatchcurves(S_CURVES , T_CURVES);

    /** Make the basis matrices different
       for each parametric direction. **/
    npatchbasis(BSPLINE, CARDINAL);

    /* Initially use the default intercept function */
    Set_Default_Routine_for_npatch();
    User_Routine(ON);

    /* Step through each intercept function */
    for(count = 0; count < 4 ; count++) {

        /** Set up an intercept function to grab
            the triangles from the surface patch. **/
        Set_User_Routine_for_npatch(intercept_functions[count]);

        /** Call the npatch function using the current
            intercept function. **/
        npatch(geomx, geomy, geomz);

        /* Display what the intercept function did. */
        swapbuffers();
        sleep(2);

        /* Clear the screen and do another one. */
        color(BLACK);
        clear();
    }
    /* clear the graphics screen and exit */
    color(BLACK);
    clear();
    gexit();
}
```

```
/*************************************************************
 This intercept function draws each triangle RED.
*************************************************************/
int intercept_function1(triangle)
float triangle[3][3];
{
    color(RED);
    poly(3, triangle);
}

/*************************************************************
 This intercept function draws each triangle YELLOW.
*************************************************************/
int intercept_function2(triangle)
float triangle[3][3];
{
    color(YELLOW);
    poly(3, triangle);
}

/*************************************************************
 This intercept function draws each triangle GREEN.
*************************************************************/
int intercept_function3(triangle)
float triangle[3][3];
{
    color(GREEN);
    poly(3, triangle);
}

/*************************************************************
 This intercept function draws each triangle BLUE.
*************************************************************/
int intercept_function4(triangle)
float triangle[3][3];
{
    color(BLUE);
    poly(3, triangle);
}
```

```
/*****************************************************************
            Program #4

   This program illustrates how the intercept function
   can be combined with an illumination model to
   provide a realistic, illuminated, three-dimensional
   solid-filled curved surface.

*****************************************************************/

#include "gl.h"     /* IRIS graphics library */

#include "device.h"
#include "basis.h"
#include "geom.h"

#define  ON        1
#define  S_CURVES  25
#define  T_CURVES  25

main()
{
    /* Declare the intercept function. */
    int  light_poly();

    /* loop variables */
    int  i,j;

    /* Initialize the graphics system. */
    ginit();
    singlebuffer();
    gconfig();
    cursoff();

    /* Clear the display. */
    color(BLACK);
    clear();

    /* Set up new viewing parameters. */
    ortho(0.0, 1023.0, 0.0, 1023.0, -1023.0, 1023.0);
    viewport(0, 1023, 0, 767);

    /* Clear drawing area. */
    color(CYAN);
    clear();

    /* Set for hidden surface elimination. */
    setdepth(0x3FFF, 0xC000);
    zclear();
    zbuffer(TRUE);
```

```
/* Load the color map ramp with a grey scale. */
for (i = 0; i <  256; i++)
{
    mapcolor(8+i, i, i, i);
}

/* Associate an id with a basis matrix. */
ndefbasis(BEZIER, beziermatrix);
ndefbasis(CARDINAL, cardinalmatrix);
ndefbasis(BSPLINE, bsplinematrix);

/** Provide a different basis for each
   parametric direction. **/
npatchbasis(CARDINAL, BEZIER);

/** Provide the number of curves in
   each parametric direction. **/
npatchcurves(S_CURVES , T_CURVES);

/* Set up the intercept function. */
Set_User_Routine_for_npatch(light_poly);
User_Routine(ON);

while(TRUE)
{
    /* Clear the z buffer. */
    zclear();

    /* Hold display if MOUSE2 is down. */
    if(getbutton(MOUSE2))
    {
        /* Resume when MOUSE1 is pressed */
        while(!getbutton(MOUSE1)) ;
    }

    /* Exit when MOUSE1 and MOUSE2 and MOUSE3 are down. */
    if(getbutton(MOUSE1) && getbutton(MOUSE2) && getbutton(MOUSE3))
        break;

    /* Clear the drawing area. */
    color(CYAN);
    clear();

    /* Set the current color. */
    color(BLACK);

    /* Draw initial surface patch. */
    npatch(geomx, geomy, geomz);
```

```
        /** Change the y coordinates to get a
            different surface patch the next
            time we draw display. **/
        for(i=1; i<3; i++)
        {
            for(j=0; j<4; j++)
            {
                geomy[i][j] = (float)(lrand48() % 900);
            }
        }

        /* Draw the surface patch. */
        npatch(geomx, geomy, geomz);

        /** Change the y coordinate values to
            make another surface patch. **/
        for(i=1; i<3; i++)
        {
            for(j=0; j<4; j++)
            {
                geomy[i][j] = (lrand48()%2)*geomy[i][j];
            }
        }
    }

    /* Clean up and exit the program. */
    color(BLACK);
    clear();
    gexit();
}


/*****************************************************************
   The user-defined intercept function used to grab the
   triangles generated in the surface patch decomposition.
*****************************************************************/
int light_poly(Triangle)
Coord Triangle[3][3];
{
    /* Put each triangle through an illumination model. */
    lightthepoly(Triangle, 3, 350.0, -1750.0, 350.0, 350.0, 1750.0, 350.0, 9, 264);
}
```

```
/*****************************************************************

                lightpoly.c

  It is a routine that computes lighting for a polygon based
  upon the angle between the Normal vector of the polygon
  and the direction to the light source.

  lightthepoly(xyz,ncoords,ax,ay,az,lx,ly,lz,colormin,colormax)

  xyz[][3] = floating coords of the polygon.

  ncoords = number of coordinates.

  ax,ay,az = interior point of the whole object. Used to determine
        outward facing normal of the polygon. This is the same
        point of reference that would be used for backface
        polygon removal.

  lx,ly.lz = vector pointing in direction of the light source.

  colormin. colormax = indices used for the colors assigned to this
                polygon. The user is responsible for setting
                up the color ramp.

  Note: the routine also puts the polygons out ordered counterclockwise
      with respect to the interior point for ease of backface polygon
      removal.

*****************************************************************/
#include <gl.h>
#include <math.h>

#define PIDIV2    1.570796327
#define CLOCKWISE 1
#define ROW       3

lightthepoly(xyz,ncoords,ax,ay,az,lx,ly,lz,colormin,colormax)
Coord       xyz[][3];
unsigned int  ncoords;
Coord       ax,ay,az;   /* interior point of the whole object. */
Coord       lx,ly,lz;   /* direction to the light source       */
int         colormin,colormax;   /* color min/max indices      */
{
  /* temp coord hold */
  Coord *txyz;

  /* loop temps */
  register unsigned short int   i,j;
```

```
/* direction test function */
int npoly_orient();

/* vectors used to compute the polygon's normal */
Coord v1[3],v2[3];

/* the polygon's normal */
Coord normal[3];

/* normal's magnitude  */
Coord normalmag;

/* light's magnitude  */
Coord lightmag;

/* dot product of N and L */
double  dotprod;

/* angle between N and L */
float radians;

/* color to use in drawing the polygon */
unsigned short int  colortouse;

/* allocate memory for a temporary array */
txyz = (Coord *) calloc((ncoords * 3), sizeof(Coord));

/** orient the polygon so that its counterclockwise with respect
   to the interior point **/
if(npoly_orient(ncoords,xyz,ax,ay,az) == CLOCKWISE)
{
  /* the polygon is clockwise, reverse it. */
  for(i=0; i < ncoords; i=i+1)
  {
    for(j=0; j < ROW; j=j+1)
    {
      *(txyz + (i * ROW) + j) = xyz[ncoords-i-1][j];
    }
  }
}
else
{
  /* no need to reverse */
  for(i=0; i < ncoords; i=i+1)
  {
    for(j=0; j < ROW; j=j+1)
    {
      *(txyz + (ROW * i) + j) = xyz[i][j];
    }
  }
}
```

```
/* the coordinates are ordered counterclockwise in array txyz */

/** compute the normal vector for the polygon using the first
   three vertices... **/

/* compute the first vector to use in the computation */
v1[0] = *(txyz + 6) - *(txyz + 3); /* txyz[2][0] - txyz[1][0] */
v1[1] = *(txyz + 7) - *(txyz + 4); /* txyz[2][1] - txyz[1][1] */
v1[2] = *(txyz + 8) - *(txyz + 5); /* txyz[2][2] - txyz[1][2] */

/* compute the second vector to use in computing the normal */
v2[0] = *(txyz    ) - *(txyz + 3); /* txyz[0][0] - txyz[1][0] */
v2[1] = *(txyz + 1) - *(txyz + 4); /* txyz[0][1] - txyz[1][1] */
v2[2] = *(txyz + 2) - *(txyz + 5); /* txyz[0][2] - txyz[1][2] */

/* the normal is v1 x v2 */
normal[0] = v1[1]*v2[2] - v1[2]*v2[1];
normal[1] = v1[2]*v2[0] - v1[0]*v2[2];
normal[2] = v1[0]*v2[1] - v1[1]*v2[0];

/* compute the magnitude of the normal */
normalmag = sqrt((normal[0]*normal[0]) +
            (normal[1]*normal[1])+
            (normal[2]*normal[2]));

/* compute the magnitude of the light */
lightmag = sqrt((lx * lx) + (ly * ly) + (lz * lz));

/* compute N . L (normal dot product with the light source direction) */
dotprod = (double) ((normal[0] * lx)
          + (normal[1] * ly)
          + (normal[2] * lz));

/* compute the unit normal */
dotprod = (double) ((dotprod/(normalmag * lightmag)));

/* dotprod = cos(theta) of the angle between N and L.
   Convert to angle in radians */
radians = acos(dotprod);
```

```
  /* compute the color we should use */
  if(-PIDIV2 <= radians && radians <= PIDIV2)
  {
    /* if the angle is negative, set to positive */
    if(radians < 0.0)
    {
      radians = -radians;
    }

    colortouse = ((colormax-colormin)/PIDIV2)*(PIDIV2-radians)+colormin;
  }
  else
  {
    colortouse = colormin;
  }

  /* set the color */
  color(colortouse);

  /* draw the poly */
  polf(ncoords,txyz);

  /* free up memory allocated for the temporary array */
  cfree(txyz, (ncoords * 3), sizeof(Coord));

}
```

```
/*****************************************************************

                npoly_orient.c

   This routine determines a polygon's orientation
   with respect to its normal and a reference point.
   Orientation is either clockwise or counter-clockwise.
   The point of reference must not lie in the polygon's
   plane.

*****************************************************************/

#include <gl.h>
#include <math.h>

int npoly_orient(ncoords,xyz,xinside,yinside,zinside)
unsigned int  ncoords;
Coord         xyz[][3];
Coord         xinside, yinside, zinside;
{
  /* loop temps */
  register unsigned short int   i,j;

  /* center coordinate of the polygon */
  Coord center[3];

  /** vector hold locations for the vectors that run
     from the center coordinate to the points of the
     polygon **/
  Coord a[3], b[3];

  /** points on line containing normal that are
     on opposite sides of the plane containing
     the polygon. **/
  Coord xn[3], xmn[3];

  /* distance to point n from pt inside. */
  float distton;

  /* distance to point -n from pt inside. */
  float disttomn;

  /* the normal vector computed from a x b */
  Coord normal[3];

  /* compute the center coordinate of the polygon */
  center[0] = 0.0;
  center[1] = 0.0;
  center[2] = 0.0;
```

```
for(i=0; i < ncoords; i++)
{
  for(j=0; j < 3; j++)
  {
    center[j] += xyz[i][j];
  }
}

/* divide out by the number of coordinates */
for(j=0; j < 3; j++)
{
  center[j] = center[j]/(float)ncoords;
}

/** check the first 2 coordinates of the
   polygon for their direction **/

/** compute vector a. It runs from the
  center coordinate to coordinate 0 **/
for(j=0; j < 3; j++)
{
  a[j] = xyz[0][j] - center[j];
}

/** compute vector b. It runs from the
  center coordinate to coordinate 1 **/
for(j=0; j <3; j++)
{
  b[j] = xyz[1][j] - center[j];
}

/* compute a x b to get the normal vector */
normal[0] = a[1]*b[2] - a[2]*b[1];
normal[1] = a[2]*b[0] - a[0]*b[2];
normal[2] = a[0]*b[1] - a[1]*b[0];

/** compute point n, offset pt from center in
   direction of normal **/
for(j=0; j < 3; j++)
{
  xn[j] = center[j] + normal[j];
}

/** compute point -n, offset pt from center
   in opposite direction from normal. **/
for(j=0; j < 3; j++)
{
  xmn[j] = center[j] - normal[j];
}
```

```
  /* compute the distance the inside pt is from point n */
  distton = sqrt((xn[0] - xinside) * (xn[0] - xinside) +
            (xn[1] - yinside) * (xn[1] - yinside) +
            (xn[2] - zinside) * (xn[2] - zinside));

  /* compute the distance the inside pt is from point -n */
  disttomn = sqrt((xmn[0] - xinside) * (xmn[0] - xinside) +
             (xmn[1] - yinside) * (xmn[1] - yinside) +
             (xmn[2] - zinside) * (xmn[2] - zinside));

  /** if the dist(n) < dist(-n), then n points back towards the
     inside point and is on the same side of the plane as inside.
     a x b is then clockwise. **/
  if(distton < disttomn)
  {
    return(1);  /* clockwise */
  }
  else
  {
    return(0);  /* counterclockwise */
  }

}
```

APPENDIX C – BENCHMARK PROGRAM

```
/*****************************************************************

  This is the BENCHMARK PROGRAM used to test the
  relative performance of the standard IRIS functions
  and the parallel routines to those standard
  functions.

*****************************************************************/

#include "gl.h"         /* IRIS graphics library */

#include "basis.h"
#include "geom.h"

#define  IRIS           /* Which set to test switch. */

#define  MAX_TIMES_THRU  100
#define  S_CURVES        25
#define  T_CURVES        25
#define  ONE             1

main()
{
     int times_thru;

    /* initialize the graphics system */
    ginit();
    doublebuffer();
    gconfig();
    cursoff();

    /* clear the graphics screen */
    color(BLACK);
    clear();

    /* set up the viewing parameters */
    ortho(0.0, 1023.0, 0.0, 1023.0, -1023.0, 1023.0);
    viewport(0, 1023, 0, 767);

    /* clear the graphics screen to CYAN */
    color(CYAN);
    clear();
```

```
#ifdef IRIS
    /* Use the standard IRIS functions */
    defbasis(BEZIER, beziermatrix);
    defbasis(CARDINAL, cardinalmatrix);
    defbasis(BSPLINE, bsplinematrix);

    patchbasis(BEZIER, BEZIER);
    patchcurves(S_CURVES , T_CURVES);
    patchprecision(ONE, ONE);
#else
    /* Use the parallel functions */
    ndefbasis(BEZIER, beziermatrix);
    ndefbasis(CARDINAL, cardinalmatrix);
    ndefbasis(BSPLINE, bsplinematrix);

    npatchbasis(BEZIER, BEZIER);
    npatchcurves(S_CURVES , T_CURVES);
    npatchprecision(ONE, ONE);
#endif
```

```
    for(times_thru = 0; times_thru < MAX_TIMES_THRU; times_thru++) {

        /* clear the graphics screen to CYAN each time thru */
        color(CYAN);
        clear();

        /* draw the wireframe surface patch in BLACK */
        color(BLACK);
#ifdef IRIS
        /* Using the IRIS function */
        patch(geomx, geomy, geomz);
#else
        /* Using the parallel function */
        npatch(geomx, geomy, geomz);
#endif
        /* display the wireframe image */
        swapbuffers();

    }

    /* clear the graphics screen and exit */
    color(BLACK);
    clear();
    gexit();

}
```

APPENDIX D – PARALLEL FUNCTIONS SOURCE CODE

```
/**************************************************************

FILE ..... npatch.c

Author ..... Gary W. TAYLOR (Captain USMC)

Date ..... 1 December 1986

Place ..... Naval Postgraduate School, Monterey CA


Environment .....

Silicon Graphics, Inc., IRIS 2400
graphics workstation, UNIX operating
system (GL2-W3.4).


Purpose .....

The following C source code provides a set
of "shadow" routines to parallel the standard
IRIS 2400 graphics workstation surface patch
routines. These parallel routines provide
their user the capability to generate
solid-filled parametric bicubic surface patches.


Notes .....

As of the current date, there are no known
side-effects or bugs associated with using
these functions.


Limitations .....

These functions can be used in immediate mode only.

**************************************************************/
```

```
#include "gl.h"     /* IRIS graphics library */
#include "stdio.h"

#define  OFF 0

/* Here is how we define a surface point. */
typedef struct {
  Coord x;
  Coord y;
  Coord z;
} Point;

/* Structure used to track user supplied basis matrices. */
static struct list_elem {

 /* Matrix id number.              */
  long  nid;

 /* Pointer to the basis matrix.     */
  float  *nmatrix;

 /* Pointer to the next basis matrix. */
  struct list_elem *next_elem_ptr;

};
```

```
/* Used in forward difference computation in the U direction. */
static  Matrix Precision_Matrix_U;

/* Used in forward difference computation in the V direction. */
static  Matrix Precision_Matrix_V;

/* The default intercept function. */
static int  _Default_system_routine ();

/* Pointer to the currently defined U basis matrix. */
static float *current_U_basis;

/* Pointer to the currently defined V basis matrix. */
static float *current_V_basis;

/* How many curves in the U direction. */
static long UCURVES;

/* How many curves in the V direction. */
static long VCURVES;

/* How may curve segments in the U direction. */
static long USEGMENTS;

/* How may curve segments in the V direction. */
static long VSEGMENTS;

/* Pointer to linked list of user supplied basis matrices.*/
static struct list_elem *head_of_list = NULL;

/* Set initial user routine to the default. */
static int  (*_User_routine) () = _Default_system_routine;

/* Initially do not call user's function. */
static int  _User_routine_is_on = OFF;
```

```
/*****************************************************************

        _Default_system_routine

 This routine is what the system automatically does if the
 user does not supply a particular intercept function.

*****************************************************************/
static int  _Default_system_routine (Triangle)
Coord Triangle[3][3];
{

/* Draw the polygon with a standard IRIS function. */
  poly (3, Triangle);

}



/*****************************************************************

        Set_User_Routine_for_npatch

 This function allows the user to supply an intercept function
 to be used to manipulate the triangles that are generated in
 the decomposition of a surface patch.

*****************************************************************/
void Set_User_Routine_for_npatch (routine)
int   (*routine) ();
{

/* Save the pointer to the user's function. */
  _User_routine = routine;

}
```

```
/*****************************************************************

            User_Routine

 This function allows the user to turn the intercept function
 on or off at will.

*****************************************************************/
void User_Routine (boolean)
int   boolean;

{

 /*
  If boolean  = 0 then the intercept function is turned OFF.
  If boolean != 0 then the intercept function is turned ON.
  */

  _User_routine_is_on = boolean;

}



/*****************************************************************

          Set_Default_Routine_for_npatch

 This function allows the user to reset the intercept routine
 to the same routine used by the system.

*****************************************************************/
void Set_Default_Routine_for_npatch ()
{

 /* Point to the default intercept function. */
  _User_routine = _Default_system_routine;

}
```

```
/*****************************************************************

                ndefbasis

 This function is equivalent to the IRIS defbasis function
 in that it allows the user to define a basis matrix for
 use in the generation of patches.  matrix is saved and
 is associated with id.  id may then be used in subsequent
 calls to npatchbasis.

*****************************************************************/
void ndefbasis (id, matrix)
long  id;
Matrix matrix;
{

 /* Special processing first time this function is called. */
 static int  has_been_called = FALSE;

 /* Data structure pointer for new entry. */
 struct list_elem *new_elem_to_add;

 /* Pointer to search linked list. */
 struct list_elem *walking_ptr;

 /* Pointer to a copy of the user supplied basis matrix. */
 float  *pmatrix;

 /* Loop variables */
 int   row;
 int   column;

 /* Get memory for the new data elements. */
 new_elem_to_add = (struct list_elem *) malloc (sizeof (struct list_elem));
 pmatrix = (float *) calloc (sizeof (Matrix), sizeof (float));

 /* Make a copy of the basis matrix passed in by the user. */
 for (row = 0; row < 4; row++)
   for (column = 0; column < 4; column++) {

     *(pmatrix + (4 * row) + column) = matrix[row][column];

   }

 /* Associate the user supplied id to this basis matrix. */
 new_elem_to_add -> nid = id;

 /* Point to the copied basis matrix. */
 new_elem_to_add -> nmatrix = pmatrix;
```

```
/** Determine how to add this information into the linked list
   of basis matrices. **/

/* Does a list already exist? */
 switch (has_been_called) {

  case TRUE:

  /* Point to the beginning of the list. */
   walking_ptr = head_of_list;

  /** Traverse the list looking to see if the id number
   already exists. **/
   while ((walking_ptr -> nid != id) &&
     (walking_ptr -> next_elem_ptr != head_of_list)) {

   /* Walk through the linked list. */
   walking_ptr = walking_ptr -> next_elem_ptr;

   }

  /* id already exists so we can reuse its allocated memory. */
   if (walking_ptr -> nid == id) {

   /* Get rid of the old basis matrix. */
   cfree (walking_ptr -> nmatrix, sizeof (Matrix), sizeof (float));

   /* Point to the replacement matrix. */
   walking_ptr -> nmatrix = pmatrix;

   /* Get rid of the un-needed data structure. */
   cfree (new_elem_to_add, 1, sizeof (struct list_elem));

   }
   else { /* The id does not exist  */

   /** Manipulate the pointers to add the new
     data element to the linked list. **/
   new_elem_to_add -> next_elem_ptr = head_of_list -> next_elem_ptr;
   head_of_list -> next_elem_ptr = new_elem_to_add;

   }

   break;
```

```
    case FALSE:

    /* No linked list of basis matrices exists so we start up one. */

    /* Create the pointer to the front of the list. */
     head_of_list = new_elem_to_add;
     head_of_list -> next_elem_ptr = head_of_list;

    /* Make sure we do not do this again. */
     has_been_called = TRUE;

     break;


    default:

     break;

  }

}
```

```
/*********************************************************************

            npatchbasis

 This function is equivalent to the IRIS patchbasis function
 in that it sets the current basis matrices for both the
 U and V parametric directions of a surface patch.  The current
 U and V bases are used when the npatch command is issued.

*********************************************************************/
int   npatchbasis (uid, vid)
long  uid, vid;
{

 struct list_elem *walking_ptr;

 /* ERROR: no linked list of basis matrices. */
 if (head_of_list == NULL) {

   fprintf (stderr, "0patchbasis: no basis matrices defined0);
   exit (-1);

 }

 /* Traverse the list looking for the desired U basis  matrix. */
 walking_ptr = head_of_list;
 while ((walking_ptr -> nid != uid) &&
    (walking_ptr -> next_elem_ptr != head_of_list)) {

   walking_ptr = walking_ptr -> next_elem_ptr;

 }

 if ((walking_ptr -> nid != uid) &&
    (walking_ptr -> next_elem_ptr == head_of_list)) {

   /* ERROR: U basis matrix does not exist in the linked list. */

   fprintf (stderr, "0patchbasis: undefined U basis matrix %d0, uid);
   exit (-1);

 }

 current_U_basis = walking_ptr -> nmatrix;
```

```
 /* Traverse the list looking for the desired V basis  matrix. */
  walking_ptr = head_of_list;
  while ((walking_ptr -> nid != vid) &&
     (walking_ptr -> next_elem_ptr != head_of_list)) {

   walking_ptr = walking_ptr -> next_elem_ptr;

  }

  if ((walking_ptr -> nid != vid) &&
     (walking_ptr -> next_elem_ptr == head_of_list)) {

    /* ERROR: V basis matrix does not exist in the linked list. */

   fprintf (stderr, "0patchbasis: undefined V basis matrix %d0, vid);
   exit (-1);

  }

  current_V_basis = walking_ptr -> nmatrix;

}
```

```
/*****************************************************************

            npatchcurves

 This function is similar to the IRIS patchcurves command.
 ucurves and vcurves set the subdivision parameters used in
 decomposing the surface patch.  An individual patch will be
 decomposed into a (ucurves -1) * (vcurves -1) grid with
 each grid generating two triangular polygons.

*****************************************************************/
void npatchcurves (ucurves, vcurves)
long  ucurves, vcurves;
{

 /* Prevent an inappropriate number of curves. */
  UCURVES = (ucurves < 2 ? 2 : ucurves - 1);
  VCURVES = (vcurves < 2 ? 2 : vcurves - 1);

 /** Set up the Precision_Matrix_U used in the forward difference
    along the U direction. **/

  Precision_Matrix_U[0][0] = 6.0 / (float) (UCURVES * UCURVES * UCURVES);
  Precision_Matrix_U[1][0] = 6.0 / (float) (UCURVES * UCURVES * UCURVES);
  Precision_Matrix_U[1][1] = 2.0 / (float) (UCURVES * UCURVES);
  Precision_Matrix_U[2][0] = 1.0 / (float) (UCURVES * UCURVES * UCURVES);
  Precision_Matrix_U[2][1] = 1.0 / (float) (UCURVES * UCURVES);
  Precision_Matrix_U[2][2] = 1.0 / (float) (UCURVES);
  Precision_Matrix_U[3][3] = 1.0;

 /** Set up the Precision_Matrix_V used in the forward difference
    along the V direction. **/

  Precision_Matrix_V[0][0] = 6.0 / (float) (VCURVES * VCURVES * VCURVES);
  Precision_Matrix_V[1][0] = 6.0 / (float) (VCURVES * VCURVES * VCURVES);
  Precision_Matrix_V[1][1] = 2.0 / (float) (VCURVES * VCURVES);
  Precision_Matrix_V[2][0] = 1.0 / (float) (VCURVES * VCURVES * VCURVES);
  Precision_Matrix_V[2][1] = 1.0 / (float) (VCURVES * VCURVES);
  Precision_Matrix_V[2][2] = 1.0 / (float) (VCURVES);
  Precision_Matrix_V[3][3] = 1.0;

}
```

```
/*****************************************************************

            npatchprecision

  This function is similar to the IRIS patchprecision routine
  but is only used to maintain consistency with the IRIS
  routines.  Its results are not used by any other function
  in the suite.

*****************************************************************/
void npatchprecision (usegments, vsegments)
long  usegments, vsegments;
{

 /* Prevent an inappropriate number of segments. */
  USEGMENTS = (usegments < 2 ? 2 : usegments - 1);
  VSEGMENTS = (vsegments < 2 ? 2 : vsegments - 1);

}
```

```
/*****************************************************************

            nspeckle

 This function finishes computation and hands off each triangle
 to an intercept function.

*****************************************************************/
static  void nspeckle (Coord_array)
Point * Coord_array[4];
{

 register  Point * Patch_array;

 /* Get enough memory to hold all the points that will be generated. */
 Patch_array = (Point *) calloc ((UCURVES + 1) * (VCURVES + 1),
                 sizeof (Point));

 {

 /* Pointer to a particular point. */
  register  Point * Where;

  register unsigned int total_points;
  register unsigned int point_count;
  register unsigned int t_count;
  register unsigned int count;
  register unsigned int Row;
  register unsigned int Column;

 /* Used in generating points on the surface. */
  Matrix control_matrix;

 /* Intermediate matrix to hold mathematical results. */
  Matrix inter1;

 /* For every point in the the U parametric direction. */
  for (point_count = 0, total_points = 0;
     point_count <= UCURVES; point_count++) {

  /* Build a control matrix for the current curve. */
   for (count = 0; count < 4; count++) {

    Where = (Point *) (Coord_array[count] + point_count);

    control_matrix[count][0] = Where -> x;
    control_matrix[count][1] = Where -> y;
    control_matrix[count][2] = Where -> z;

    }
```

```
  /** Generate the matrix to compute the forward difference on.
    The forward difference matrix is equal to
    Precision_Matrix_V x current_V_basis x control_matrix. **/

   pushmatrix ();
   loadmatrix (control_matrix);
   multmatrix (current_V_basis);
   multmatrix (Precision_Matrix_V);
   getmatrix (inter1);
   popmatrix ();

  /* Generate the points on the curve in the V direction */
   for (t_count = 0; t_count <= VCURVES; t_count++, total_points++) {

   (Patch_array + total_points) -> x = inter1[3][0];
   (Patch_array + total_points) -> y = inter1[3][1];
   (Patch_array + total_points) -> z = inter1[3][2];

   /* Do the forward difference. */
   for (Row = 3; Row > 0; Row--)
    for (Column = 0; Column < 4; Column++)
     inter1[Row][Column] = inter1[Row][Column] + inter1[Row - 1][Column];

   }

  }

}
```

```
{

  /* Place to put a triangle to send out to user */
  Coord Triangle_1[4][3];

  register Point *Where;
  register unsigned int Row;
  register unsigned int Column;

 /* Decompose the patch into its individual triangles. */
  for (Row = 0; Row < UCURVES; Row++)
    for (Column = 0; Column < VCURVES; Column++) {

     Where = (Patch_array + ((VCURVES + 1) * Row) + Column);

     Triangle_1[0][0] = Where -> x;
     Triangle_1[0][1] = Where -> y;
     Triangle_1[0][2] = Where -> z;

     Triangle_1[1][0] = (Where + 1) -> x;
     Triangle_1[1][1] = (Where + 1) -> y;
     Triangle_1[1][2] = (Where + 1) -> z:

     Triangle_1[2][0] = (Where + VCURVES + 1) -> x;
     Triangle_1[2][1] = (Where + VCURVES + 1) -> y;
     Triangle_1[2][2] = (Where + VCURVES + 1) -> z;

     Triangle_1[3][0] = (Where + VCURVES + 2) -> x;
     Triangle_1[3][1] = (Where + VCURVES + 2) -> y;
     Triangle_1[3][2] = (Where + VCURVES + 2) -> z;

     /* Does the user have an intercept routine? */
     if (_User_routine_is_on) {
       /* Yes */
      (*_User_routine) (&Triangle_1[0][0]);
      (*_User_routine) (&Triangle_1[1][0]);
     }
     else {
       /* No user routine, so we use the default. */
      _Default_system_routine (&Triangle_1[0][0]);
      _Default_system_routine (&Triangle_1[1][0]);
     }
     }
 }

 /* Return the memory used to the system. */
 cfree (Patch_array, (UCURVES + 1) * (VCURVES + 1), sizeof (Point));
}
```

```
/*************************************************************************

                npatch

 This function rearranges the input matrices into a form readily
 used in computing points along the four curves defined by those
 matrices.  It then computes points for each curve in the U
 direction using the technique of forwards differencing of a
 matrix.  Using these points we can then generate points along
 the a curve in the V direction.

*************************************************************************/
void npatch (geomx, geomy, geomz)
Coord geomx[4][4], geomy[4][4], geomz[4][4];
{

 register  Point * Coord_array[4];

 /* One control matrix for each curve. */
  Matrix ctrl_pts1;
  Matrix ctrl_pts2;
  Matrix ctrl_pts3;
  Matrix ctrl_pts4;
```

```
/* Load the appropriate control matrix for each curve. */

/* Curve 1 */
ctrl_pts1[0][0] = geomx[0][0];
ctrl_pts1[0][1] = geomy[0][0];
ctrl_pts1[0][2] = geomz[0][0];

ctrl_pts1[1][0] = geomx[1][0];
ctrl_pts1[1][1] = geomy[1][0];
ctrl_pts1[1][2] = geomz[1][0];

ctrl_pts1[2][0] = geomx[2][0];
ctrl_pts1[2][1] = geomy[2][0];
ctrl_pts1[2][2] = geomz[2][0];

ctrl_pts1[3][0] = geomx[3][0];
ctrl_pts1[3][1] = geomy[3][0];
ctrl_pts1[3][2] = geomz[3][0];

/* Curve 2 */
ctrl_pts2[0][0] = geomx[0][1];
ctrl_pts2[0][1] = geomy[0][1];
ctrl_pts2[0][2] = geomz[0][1];

ctrl_pts2[1][0] = geomx[1][1];
ctrl_pts2[1][1] = geomy[1][1];
ctrl_pts2[1][2] = geomz[1][1];

ctrl_pts2[2][0] = geomx[2][1];
ctrl_pts2[2][1] = geomy[2][1];
ctrl_pts2[2][2] = geomz[2][1];

ctrl_pts2[3][0] = geomx[3][1];
ctrl_pts2[3][1] = geomy[3][1];
ctrl_pts2[3][2] = geomz[3][1];
```

```
/* Curve 3 */
ctrl_pts3[0][0] = geomx[0][2];
ctrl_pts3[0][1] = geomy[0][2];
ctrl_pts3[0][2] = geomz[0][2];

ctrl_pts3[1][0] = geomx[1][2];
ctrl_pts3[1][1] = geomy[1][2];
ctrl_pts3[1][2] = geomz[1][2];

ctrl_pts3[2][0] = geomx[2][2];
ctrl_pts3[2][1] = geomy[2][2];
ctrl_pts3[2][2] = geomz[2][2];

ctrl_pts3[3][0] = geomx[3][2];
ctrl_pts3[3][1] = geomy[3][2];
ctrl_pts3[3][2] = geomz[3][2];

/* Curve 4 */
ctrl_pts4[0][0] = geomx[0][3];
ctrl_pts4[0][1] = geomy[0][3];
ctrl_pts4[0][2] = geomz[0][3];

ctrl_pts4[1][0] = geomx[1][3];
ctrl_pts4[1][1] = geomy[1][3];
ctrl_pts4[1][2] = geomz[1][3];

ctrl_pts4[2][0] = geomx[2][3];
ctrl_pts4[2][1] = geomy[2][3];
ctrl_pts4[2][2] = geomz[2][3];

ctrl_pts4[3][0] = geomx[3][3];
ctrl_pts4[3][1] = geomy[3][3];
ctrl_pts4[3][2] = geomz[3][3];
```

```
{

 register  Point *Where;

 register unsigned int Row;
 register unsigned int Column;
 register unsigned int point_count;
 register unsigned int count;

 /* An array of pointers to our control matrices. */
 float  *matrix_pointer[4];

 /* Matrix used to hold mathematical results. */
 Matrix inter1;

/* Initialize the array */
 matrix_pointer[0] = (float *) ctrl_pts1;
 matrix_pointer[1] = (float *) ctrl_pts2;
 matrix_pointer[2] = (float *) ctrl_pts3;
 matrix_pointer[3] = (float *) ctrl_pts4;

/* Get enough memory to hold the points. */
 Coord_array[0] = (Point *) calloc (UCURVES + 1, sizeof (Point));
 Coord_array[1] = (Point *) calloc (UCURVES + 1, sizeof (Point));
 Coord_array[2] = (Point *) calloc (UCURVES + 1, sizeof (Point));
 Coord_array[3] = (Point *) calloc (UCURVES + 1, sizeof (Point));

/* For each curve. */
 for (count = 0; count < 4; count++) {

 /* Generate the matrix used in the forward difference for this curve. */
  pushmatrix ();
  loadmatrix (matrix_pointer[count]);
  multmatrix (current_U_basis);
  multmatrix (Precision_Matrix_U);
  getmatrix (inter1);
  popmatrix ();
```

```
    /* For each curve generate points in the U parametric direction. */
     for (point_count = 0; point_count <= UCURVES; point_count++) {

      Where = (Point *) (Coord_array[count] + point_count);

      Where -> x = inter1[3][0];
      Where -> y = inter1[3][1];
      Where -> z = inter1[3][2];

      /* Do the forward difference. */
      for (Row = 3; Row > 0; Row--)
       for (Column = 0; Column < 4; Column++)
        inter1[Row][Column] = inter1[Row][Column] + inter1[Row - 1][Column];

     }

    }

   }

  /* Call function to finish computations and display. */
  nspeckle (Coord_array);

  /* Return the memory used back to the system. */
  cfree (Coord_array[0], (UCURVES + 1), sizeof (Point));
  cfree (Coord_array[1], (UCURVES + 1), sizeof (Point));
  cfree (Coord_array[2], (UCURVES + 1), sizeof (Point));
  cfree (Coord_array[3], (UCURVES + 1), sizeof (Point));

}
```

LIST OF REFERENCES

1. Mortenson, Michael E., *Geometric Modeling,* John Wiley and Sons, 1985.

2. Foley, James D. and Van Dam, Andries, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, 1982.

3. Giloi, Wolfgang, *Interactive Computer Graphics - Data Structures, Algorithms, Languages,* Prentice-Hall, Inc., 1978.

4. Silicon Graphics Incorporation, "IRIS User's Guide Version 2.1," Document Number 5001-051-001-1, 1985.

5 Schmitt, Francis J. M., Barsky, Brian A., and Du, Wen-Hui, *An Adaptive Subdivision Method for Surface-fitting,* paper presented at the ACM SIGGRAPH86, Dallas, Texas, 18-22 August 1986.

Distribution List

Defense Technical Information Center, Cameron Station, Alexandria, VA 22314	2 copies
Library, Code 0142, Naval Postgraduate School, Monterey, CA 93943	2 copies
Center for Naval Analyses, 2000 N. Beauregard Street, Alexandria, VA 22311	1 copy
Director of Research Administration, Code 012, Naval Postgraduate School, Monterey, CA 93943	1 copy
Roger Casey, Naval Ocean Systems Center, Code 854, San Diego, CA 92152	1 copy
Mr. Russell Davis, HQ, USACDEC, Attention: ATEC-IM, Fort Ord, CA 93941	1 copy
Prof. Michael Zyda, Code 52Zk, Naval Postgraduate School, Monterey, CA 93943	152 copies